(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 1 380 946 A2

(12) # EUROPEAN PATENT APPLICATION

(43) Date of publication:
14.01.2004 Bulletin 2004/03

(51) Int Cl.7: G06F 9/45

(21) Application number: 03020509.0

(22) Date of filing: 11.10.1999

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE

(30) Priority: 10.10.1998 GB 9822075
14.01.1999 US 115952 P

(62) Document number(s) of the earlier application(s) in accordance with Art. 76 EPC:
99949129.3 / 1 119 807

(71) Applicant: Transitive Limited
London (GB)

(72) Inventors:
- Rawsthorne, Alasdair
  Hanging Ditch Manchester M4 3TR (GB)
- Souloglou, Jason
  Hanging Ditch Manchester M4 3TR (GB)

(74) Representative: Robinson, Ian Michael
Appleyard Lees,
15 Clare Road
Halifax HX1 2HY (GB)

Remarks:
This application was filed on 16 - 09 - 2003 as a divisional application to the application mentioned under INID code 62.

(54) **Program code conversion with reduced translation**

(57) In one aspect, multiple blocks of intermediate representation (2a,2b) are permitted derived from a single portion of program code. Each of the multiple blocks represent the portion of program code under different entry conditions (e.g. for a different status of a processor register d0). In many cases only relatively few blocks (2a, 2b) will be required, and other potential variants of the portion of program code are never encountered. A second aspect of the invention applies to individual program code instructions (2) which have different effects or functions at different iterations. Corresponding special-case intermediate representation (2a,2b) is generated representing only the functionality of the instruction that is required at a particular iteration.

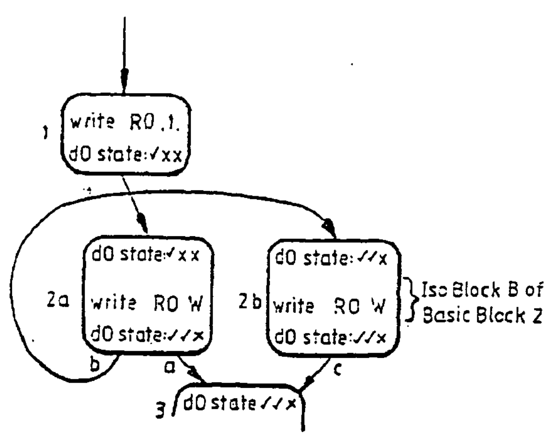

Figure 2

EP 1 380 946 A2

Description

[0001] The present invention relates to a method and system for converting program code from one format to another. In particular, the invention relates to a method and system for providing an intermediate representation of a computer program or a Basic Block of a program (a Basic Block of a program is a block of instructions that has only one entry point, at a first instruction, and only one exit point, at a last instruction of the block). For instance, the present invention provides a method and system for the translation of a computer program which was written for one processor so that the program may run efficiently on a different processor; the translation utilising an intermediate representation and being conducted in a block by block mode.

[0002] Intermediate representation is a term widely used in the computer industry to refer to forms of abstract computer language in which a program may be expressed, but which is not specific to, and is not intended to be directly executed on, any particular processor. Intermediate representation is for instance generally created to allow optimisation of a program. A compiler for example will translate a high level language computer program into intermediate representation, optimise the program by applying various optimisation techniques to the intermediate representation, then translate the optimised intermediate representation into executable binary code. Intermediate representation is also used to allow programs to be sent across the Internet in a form which is not specific to any processor. Sun Microsystems have for example developed a form of intermediate representation for this purpose which is known as bytecode. Bytecode may be interrupted on any processor on which the well known Java (trade mark) run time system is employed.

[0003] Intermediate representation is also commonly used by emulation systems which employ binary translation. Emulation systems of this type take software code which has been compiled for a given processor type, convert it into an intermediate representation, optimise the intermediate representation, then convert the intermediate representation into a code which is able to run on another processor type. Optimisation of generating an intermediate representation is a known procedure used to minimise the amount of code required to execute an emulated program. A variety of known methods exist for the optimisation of an intermediate representation.

[0004] An example of a known emulation system which uses an intermediate representation for performing binary translation is the FlashPort system operated by AT&T. A customer provides AT&T with a program which is to be translated (the program having been compiled to run on a processor of a first type). The program is translated by AT&T into an intermediate representation, and the intermediate representation is optimised via the application of automatic optimisation routines, with the assistance of technicians who provide input when the optimisation routines fail. The optimised intermediate is then translated by AT&T into code which is able to run on a processor of the desired type. This type of binary translation in which an entire program is translated before it is executed is referred to as "static" binary translation. Translation times can be anything up to several months.

[0005] In an alternative form of emulation, a program in code of a subject processor (i.e. a first type of processor for which the code is written and which is to be emulated) is translated dynamically in Basic Blocks, via an intermediate representation, into code of a target processor (i.e. a second type of processor on which the emulation is performed).

[0006] Afzal T et al: 'Motorola PowerPC Migration Tools-Emulation and Translation' Digest of Papers of the Computer Society Computer Conference Compcon, US, Los Alamitos, IEEE Comp.SOC.Press, vol CONF. 41, 25-28 February 1996, pages 145-150, ISBN: 0-8186-7414-8. This paper describes emulation and translation methods for transferring existing applications to the Motorola PowerPC architecture.

[0007] According to the present invention there is provided an apparatus and method as set forth in the appended claims. Preferred features of the invention will be apparent from the dependent claims, and the description which follows.

[0008] An aim of the present invention is to provide a method of generating intermediate representation that reduces the amount of translated or optimised code.

[0009] In one aspect of the present invention there is provided a method of generating an intermediate representation of computer program code, the method comprising the computer implemented steps of:

> on the initial translation of a given portion of subject code, generating and storing only intermediate representation which is required to execute that portion of program code with a prevailing set of conditions; and
>
> whenever subsequently the same portion of subject code is entered, determining whether intermediate representation has previously been generated and stored for that portion of subject code for the subsequent conditions, and if no such intermediate representation has previously been generated, generating additional intermediate representation required to execute said portion of subject code with said subsequent conditions.

[0010] The present invention reduces the amount of translated code by permitting multiple, but simpler, blocks of intermediate representation code for single Basic Blocks of subject code. In most cases only one simpler translated block will be required.

[0011] Also according to the present invention there

is provided a method of generating an intermediate representation of computer code written for running on a programmable machine, said method comprising:

(i) generating a plurality of register objects for holding variable values to be generated by the program code; and
(ii) generating a plurality of expression objects representing fixed values and/or relationships between said fixed values and said variable values according to said program code;

said intermediate representation being generated and stored for a block of computer code and subsequently re-used if the same block of code is later re-entered, and wherein at least one block of said first computer program code can have alternative un-used entry conditions or effects of functions and said intermediate representation is only initially generated and stored as required to execute that block of the program code with a then prevailing set of conditions.

[0012] For instance, in a preferred embodiment of the invention the method includes computer implemented steps of:

generating an Intermediate Representation Block (IR Block) of intermediate representation for each Basic Block of the program code as it is required by the program, each IR Block representing a respective Basic Block of program code for a particular entry condition;

storing target code corresponding to each IR Block; and

when the program requires execution of a Basic Block for a given entry condition, either:

a) if there is a stored target code representing that Basic Block for that given entry condition, using said stored target code; or

b) if there is no stored target code representing that Basic Block for that given entry condition, generating a further IR Block representative of that Basic Block for that given entry condition.

[0013] A Basic Block is a group of sequential instructions in the subject processor i.e. subject code. A Basic Block has only one entry point and terminates either immediately prior to another Basic Block or at a jump, call or branch instruction (whether conditional or unconditional). An IR Block is a block of intermediate representation and represents the translation of a Basic Block of subject code. Where a set of IR Blocks have been generated to represent the same Basic Block but for different entry conditions, the IR Blocks within that set are referred to below as IsoBlocks.

[0014] It is a property of subject code that:

i) a Basic Block of code may have alternative and unused entry conditions. This may be detected at the time the translation is performed; and

ii) a Basic Block of code may have alternative, and unused, possible effects or functions. In general, this will only be detectable when the translated code is executed.

[0015] This aspect of the invention may be applied to static translation, but is particularly applicable to emulation via dynamic binary translation. According to the invention, an emulation system may be configured to translate a subject processor program Basic Block by Basic Block. When this approach is used, the state of an emulated processor following execution of a Basic Block of program determines the form of the IR Block used to represent a succeeding Basic Block of the program.

[0016] In contrast, in known emulators which utilise translation, an intermediate representation of a Basic Block of a program is generated, which is independent of the entry conditions at the beginning of that Basic Block of program. The intermediate representation is thus required to take a general form, and will include for example a test to determine the validity (or otherwise) of abstract registers. In contrast to this, in the present invention the validity (or otherwise) of the abstract registers is already known and the IR block therefore does not need to include the validity test. Furthermore, since the validity of the abstract registers is known, the IR block will include only that code which is required to combine valid abstract registers and is not required to include code capable of combining all abstract registers. This provides a significant performance advantage, since the amount of code required to be translated into intermediate representation for execution is reduced. If a Basic Block of a program has previously been translated into intermediate representation for a given set of entry conditions, and if it commences with different entry conditions, the same Basic Block of the program will be re-translated into an IsoBlock of intermediate representation.

[0017] A further advantage of the invention is that the resulting IR Blocks and IsoBlocks of intermediate representation are less complex than an intermediate representation which is capable of representing all entry conditions, and may therefore be optimised more quickly and will also be translated into target processor code which executes more quickly.

[0018] The present invention also exploits subject code instructions which may have a number of possible effects or functions, not all of which may be required when the instruction is first executed, and some of which may not in fact be required at all. This aspect of the invention may only be used when the intermediate repre-

sentation is generated dynamically. That is, the method according to the present invention preferably comprises, when the intermediate representation of the program is generated dynamically as the program is running, the computer implemented steps of:

at a first iteration of a particular subject code instruction having a plurality of possible effects or functions, generating and storing special-case intermediate representation representing only the specific functionality required at that iteration; and

at each subsequent iteration of the same subject code instruction, determining whether special-case intermediate representation has been generated for the functionality required at said subsequent iteration and generating additional special-case intermediate representation specific to that functionality if no such special-case intermediate representation has previously been generated.

[0019] This aspect of the invention overcomes a problem associated with emulation systems, namely the translation of unnecessary features of subject processor code. When a complex instruction is decoded from a subject processor code into the intermediate representation, it is common that only a subset of the possible effects of that instruction will ever be used at a given place in the subject processor program. For example, in a CISC (Complex Instruction Set Computer) instruction set, a memory load instruction may be defined to operate differently depending on what type of descriptor is contained in a base register (the descriptor describes how information is stored in the memory). However, in most programs only one descriptor type will be used by each individual load instruction of that program. A translator in accordance with this invention will generate special-case intermediate representation which includes a load instruction defined for only that descriptor type.

[0020] Preferably, when the special-case intermediate representation is generated and stored an associated test procedure is generated and stored to determine on subsequent iterations of the respective subject code instruction whether the required functionality is the same as that represented by the associated stored special-case intermediate representation, and where additional special-case intermediate representation is required an additional test procedure associated with that special-case intermediate representation is generated and stored with that additional special-case intermediate representation.

[0021] Preferably, the additional special case intermediate representation for a particular subject code instruction and the additional associated test procedure is stored at least initially in subordinate relation to any existing special-case intermediate representation and associated test procedures stored to represent the same subject instruction, such that upon the second and subsequent iteration of a subject code instruction determination of whether or not required special-case intermediate representation has previously been generated is made by performing said test procedures in the order in which they were generated and stored until either it is determined that special-case intermediate representation of the required functionality exists or it is determined that no such required special-case intermediate representation exists in which case more additional intermediate representation and another associated test procedure is generated.

[0022] Preferably, the intermediate representation is optimised by adjusting the ordering of the test procedures such that test procedures associated with more frequently used special-case intermediate representation are run before test procedures associated with less frequently used special-case intermediate representation rather than ordering the test procedures in the order in which they are generated.

[0023] Intermediate representation generated in accordance with any of the above methods may be used, for instance, in the translation of a computer program written for execution by a processor of a first type so that the program may be executed by a different processor, and also as a step in optimising a computer program. In the latter case, intermediate representation may be generated to represent a computer program written for execution by a particular processor, that intermediate representation may then be optimised and then converted back into the code executable by that same processor.

[0024] Although the invention as described above relates to the generation of intermediate representation, the steps described therein may be applied to the generation of target code directly from subject code, without the generation of intermediate representation.

[0025] Thus, the present invention also provides a method of generating target code representation of computer program code, the method comprising the computer implemented steps of:

on the initial translation of a given portion of subject code, generating and storing only target code which is required to execute that portion of program code with a prevailing set of conditions; and

whenever subsequently the same portion of subject code is entered, determining whether target code has previously been generated and stored for that portion of subject code for the subsequent conditions, and if no such target code has previously been generated, generating additional target code required to execute said portion of subject code with said subsequent conditions.

[0026] It will be appreciated that many of the features and advantages described in relation to the generation of intermediate representation will correspondingly apply to the generation of target code.

[0027] A specific embodiment of the present invention applied to a dynamic emulation system will now be described, by way of example only, with reference to the accompanying drawings, in which:

Figures 1 and 2 are schematic illustrations of the manner in which a dynamic emulation system generates an intermediate representation of a Basic Block of a program which depends upon starting conditions at the beginning of that Basic Block of the program.

[0028] The embodiment of the invention described below is a system for emulating the instruction set of one processor on a processor of a different type. In the following description the term subject processor refers to a processor which is to be emulated by an emulation system, and target processor refers to a processor upon which the emulation system is run. The system is a dynamic binary translation system which essentially operates by translating Basic Blocks of instructions in the subject processor code into target processor code as they are required for execution. The emulation system, as described below, comprises three major components, referred to respectively as a Front End, a Core, and a Back End. The subject processor instructions are decoded and converted into the intermediate representation by the Front End of the emulation system. The Core of the emulation system analyses and optimises the intermediate representation of the subject processor instructions, and the Back End converts the intermediate representation into target processor code which will run on the target processor.

[0029] The Front End of the system is specific to the subject processor that is being emulated. The Front End configures the emulation system in response to the form of subject processor, for example specifying the number and names of subject processor registers which are required by the emulation, and specifying to the Back End the virtual memory mappings that will be required.

[0030] Subject processor instructions are converted into intermediate representation in Basic Blocks, each resulting intermediate representation block (IR Block) then being treated as a unit by the Core for emulation, caching, and optimisation purposes.

[0031] The Core optimises the intermediate representation generated by the Front End. The Core has a standard form irrespective of the subject and target processors connected to the emulation system. Some Core resources however, particularly register numbers and naming, and the detailed nature of IR Blocks, are configured by an individual Front End to suit the requirements of that specific subject processor architecture.

[0032] The Back End is specific to the target processor and is invoked by the Core to translate intermediate representation into target processor instructions. The Back End is responsible for allocating and managing target processor registers, for generating appropriate memory load and store instructions to emulate the subject processor correctly, for implementing a calling sequence to permit the Core to call dynamic routines, and to enable those dynamic routines to call Back End and Front End routines as appropriate.

[0033] The operation of the emulation system will now be described in more detail. The system is initialised, to create appropriate linkages between Front End, Core, and Back End. At the end of initialisation, an execution cycle is commenced, and the Core calls the front End to decode a first Basic Block of subject processor instructions. The Front End operates instruction by instruction, decoding each subject processor instruction of the Basic Block in turn, and calling Core routines to create an intermediate representation for each sub-operation of each instruction. When the Front End decodes an instruction that could possible cause a change of program sequence (for instance a jump, call or branch instruction, whether conditional or unconditional), it returns to the Core before decoding further subject processor instructions (thereby ending that Basic Block of code).

[0034] When the Front End has translated a Basic Block of subject processor instructions into the intermediate representation, the Core optimises the intermediate representation then invokes the Back End to dynamically generate a sequence of instructions in the target processor code (target instructions) which implement the intermediate representation of the Basic Block. When that sequence of target instructions is generated it is executed immediately. The sequence of target processor instructions is retained in a cache for subsequent reuse (unless it is first overwritten).

[0035] When the target processor instructions have been executed a value is returned which indicates an address which is to be executed next. In other words, the target processor code evaluates any branch, call or jump instructions, whether conditional or unconditional, at the end of the Basic Block, and returns its effect. This process of translation and execution of Basic Blocks continues until a Basic Block is encountered which has already been translated.

[0036] When target code representing the next Basic Block has been used previously and has been stored in the cache, the Core simply calls that target code. When the end of the Basic Block is reached, again the target code supplies the address of the next subject instruction to be executed, and the cycle continues.

[0037] Both the intermediate representation and target-processor code are linked to Basic Blocks of subject processor instructions. The intermediate representation is linked so that the optimiser can generate efficient emulations of groups of frequently-executed IR Blocks, and the target code is linked so that the second and subsequent executions of the same Basic Block can execute the target code directly, without incurring the overhead of decoding the instructions again.

[0038] The Front End requests that a required number

of abstract registers be defined in the Core at initialisation time. These abstract registers (labelled Ri) represent the physical registers that would be used by the subject processor instructions if they were to run on a subject processor. The abstract registers define the state of the subject processor which is being emulated, by representing the expected effect of the instructions on the subject processor registers.

[0039] The intermediate representation represents the subject processor program by assigning expression objects to abstract registers. Expression objects are a means of representing in the intermediate representation the effect of, for example, an individual arithmetic, logical, or conditional operation. Since many subject processor instructions carry out manipulation of data, most instructions generate expression objects to represent their individual sub-operations. Expression objects are used, for example, to represent addition operations, condition setting operations, conditional evaluation in conditional branches, and memory read operations. The abstract registers are referenced to expression objects, which are referenced to other expression objects so that each Basic Block of subject processor instructions is represented by a number of inter-referenced expression objects which may be considered as an expression forest.

[0040] Optimisation of the intermediate generalisation may be achieved by eliminating redundant lines of subject processor code, as described below.

[0041] When a complicated instruction is decoded from the subject processor code into intermediate representation, it is common that only a subset of the possible effects of that instruction will ever be used at a given place in the subject program. For example, in a CISC instruction set, a memory load instruction may be defined to operate differently depending on what type of descriptor is contained in a base register (the descriptor describes how information is stored in the memory). However, in most programs only one descriptor type will be used by each individual load instruction in the program.

[0042] In the emulation system of the invention, the Front End queries run-time values as the subject processor program is being executed, and generates special-case intermediate representation as necessary. In the example given above, special-case intermediate representation will be generated which omits those parts of the memory load instruction which relate to descriptor types not used by the program.

[0043] The special-case is guarded by a test which, if is ever detects at run-time that additional functionality is required, causes re-entry to the Front End to produce additional code. If, during optimisation, it is discovered that an initial assumption is wrong (for example an assumption that a particular descriptor type is being used throughout the program), the optimiser will reverse the sense of the test, so that a more frequently-used functionality will be selected more quickly than the initially chosen, less frequently-used functionality.

[0044] The emulation system described herein is capable of emulating subject processors which use variable-sized registers, as described below. This description is helpful as an example of the entry conditions that may be encountered in practice for each Basic Block of program code.

[0045] An example of an instruction-set architecture which uses a variable-sized register is the architecture of the Motorola 68000 series of processor. In the 68000 architecture, instructions that are specified as 'long' (.1) operate on all 32 bits of a register or memory location. Instructions that are specified as 'word' (.w) or 'byte' (.b) operate on only the bottom 16 and bottom 8 bits respectively, of a 32-bit register or memory location. Even if a byte addition, for example, generates a carry, that carry is not propagated into the 9th bit of the register.

[0046] To avoid conflict between different instructions operating on data of different widths (in this example in a 68000 processor), for each subject processor register the system according to the invention creates a set of three abstract registers, each register of the set being dedicated to data of a given width (i.e. one register for each of byte, word and long word data). Each register of a 68000 processor always stores a 32-bit datum, whereas instructions may operate on 8-bit or 16-bit subsets of this 32-bit datum. In the Core of a system whose Front End is configured to be connected to a 68000, byte values for a subject processor 'd0', for example, will be stored in an abstract register labelled 'D0_B', whereas word values are stored in a separate abstract register labelled 'D0_W', and long values are stored in a third abstract register labelled 'D0_L'. In contrast to the data registers, the 68000 address registers have only two valid address sizes: word and long. In this example therefore, the Core will need only two abstract registers to represent each 68000 address register: 'A0_L' and 'A0_W'.

[0047] If no conflict regarding instruction size arises within a particular Basic Block of subject processor instructions (i.e. if all of the instructions within that Basic Block are of the same bit width), the data contained in the appropriate abstract register can be accessed freely. If, however, a conflict does arise (i.e. instructions of different bit widths are stored/read from a given subject processor register), the correct data may be derived by combining the contents of two or more abstract registers in an appropriate way. An advantage of this scheme is that the Core is simplified since all operations on abstract registers are carried out on 32-bit data items.

[0048] The difference between subject processor registers and abstract registers is of importance when considering the effect of variable-sized registers. A subject processor register, such as 'd0' in the 68000 architecture, is a unit of fast store in a subject processor, which unit is referred to in assembler operands by its label ('d0' in this case). In contrast to this, abstract registers are objects which form an integral part of the intermediate

representation of the Core, and are used to represent the set of subject processor registers. Abstract registers contain extra semantics over and above those in a subject processor register, and any number of abstract registers may be used to represent a single subject processor register, provided that the correct semantics for interaction with the subject processor are preserved. As mentioned above, in the invention, the Front End requires three abstract registers to represent each 68000 data register (i.e. one for each width of data: byte, word and long word), and two abstract registers to represent each 68000 address register. In contrast to this, an implementation of a MIPS Front End, for example, might map a single subject processor register to a single abstract register.

[0049] It is desirable that the unambiguous current state of each subject processor register is known at all times, so that the correct combination of abstract registers may be made when a read instruction is made to the subject processor register which those abstract registers represent.

[0050] If the initial state of a subject processor register on entry to a Basic Block were to be unknown at translate time, target-processor code to test the state of the register would have to be generated. For this reason, the emulation system preferably ensures that the state of each subject processor register is always known at translate time. In the preferred system according to the present invention this is done by propagating the register state from one Intermediate Representation (IR) Block to the next. For example, IR Block 1 propagates the state of 'd0' to its successor IR Block 2, and IR Block 2 acts in a similar way propagating register state to IR Block 3. An example of this propagation of the subject processor register state is shown in Figure 1.

[0051] In Figure 1, IR Block 2 has two possible successors, either IR Block 3 or back at the beginning of IR Block 2. The route between IR Blocks 2 and 3 is shown with an arrow labelled as 'a'. The route from the end back to the beginning of IR Block 2 is shown as a dotted line labelled 'b' (a dotted line is used since, although this route exists it has not yet been traversed in the current execution of the translated program). If during the execution of the translated program, IR Block 2 were to branch back to itself along route 'b', the states it propagates would be incompatible with the abstract register states which were originally passed to IR Block 2 by IR Block 1. Since the intermediate representation is specific to the state of the abstract registers IT Block 2 cannot be re-executed. For the correct operation of the invention across IR Block boundaries, each IR Block must have an unambiguous representation of the current state of the subject processor register (as represented by the abstract registers). The existence of route 'b' therefore is incompatible with the operation of the invention across the boundary between IR Block 1 and IR Block 2.

[0052] To overcome this problem the invention is able to represent a Basic Block of subject processor code using more than one IR Block with different entry conditions. The IR Blocks which are used to represent a single Basic Block with different entry conditions are referred to as IsoBlocks. Each IsoBlock is a representation of the same Basic Block of subject processor code, but under different entry conditions. Figure 2 shows two IsoBlocks which are used to overcome the problem illustrated in Figure 1. IsoBlock 2a is a correct representation of Basic Block 2, but only if the state of subject processor register 'd0' at the start of IR Block 2 is ✓XX (this corresponds to IR block 2 of Figure 1). When successor route 'b' in Figure 2 is traversed for the first time, all the IsoBlocks in existence which represent Basic Block 2, (there is only one in this case, the IR Block), are tested for compatibility with the abstract register states that are to be propagated (i.e. ✓ ✓X). If a compatible IsoBlock is found (i.e. one that begins with the register state ✓ ✓X), the successor route 'b' will be permanently connected to that IsoBlock. In the illustrated example of Figure 2 there is no existing IsoBlock that route 'b' is compatible with, and so new IsoBlock 2b, must be created. IsoBlock 2b is created by decoding for a second time the subject processor instructions that make up Basic Block 2, using an initial assumption that the state of subject processor register 'd0' at the start of Basic Block 2 is ✓ ✓ X.

[0053] When successor route 'c', originating from IsoBlock 2b, is traversed for the first time, a compatibility test is performed with IR Block 3. Since route 'c' is compatible with IR Block 3, a new IsoBlock does not need to be created, and both successor route 'a' and successor route 'c' are connected to IR Block 3.

[0054] The low-level details concerning the compatibility test mentioned above will differ between Front End modules, since they depend on the exact nature of overlapping registers provided in the subject processor architecture. The necessary modifications of these details will be apparent to those skilled in the art.

[0055] The principle of creating an IsoBlock of intermediate representation for a given set of abstract register states on entry may be widened to an intermediate representation which represents a Basic Block of subject processor code for specific values of a broad set of initial conditions. Known intermediate representations represent a block of instructions for all possible initial starting conditions, and are therefore required to include a significant amount of flexibility. Intermediate representation formed in this manner is by necessity complicated, and will in general include elements which will never be used during execution.

[0056] The intermediate representation according to the invention is advantageous because it represents a Basic Block of code for specific values of entry conditions and is therefore more compact than known intermediate representations. A further advantage of the invention is that all intermediate representation which is generated is used at least once, and time is not wasted

producing unnecessary additional representation.

[0057] The above description is directed towards emulation, but it will be appreciated by those skilled in the art that the invention may also be used in other applications, for example the optimisation of code during compilation.

[0058] Although a few preferred embodiments have been shown and described, it will be appreciated by those skilled in the art that various changes and modifications might be made without departing from the scope of the invention, as defined in the appended claims.

Claims

1. A method of generating an intermediate representation of computer program code, the method comprising the steps of:

   on an initial translation of a first portion of program code (2), generating and storing only intermediate representation (2a) which is required to execute that first portion of program code with a first prevailing set of conditions (d0_state: ✓XX); and
   whenever subsequently the first portion of program code (2) is entered, determining whether intermediate representation (2b) has previously been generated and stored for that first portion of program code (2) for then-prevailing conditions (d0_state: ✓ ✓X), and if no such intermediate representation has previously been generated, generating additional intermediate representation (2b) required to execute said first portion of program code (2) with said then-prevailing conditions (d0_state: ✓ ✓ X).

2. The method according to claim 1, wherein the conditions are entry conditions for the first portion of program code (2).

3. The method of claim 1 or 2, comprising::

   generating an Intermediate Representation Block (IR Block) (1,2,3) of intermediate representation for each Basic Block of program code as it is required by the program, each IR Block (1,2,3) representing a respective Basic Block of the program code for a particular entry condition;

   storing target code corresponding to each IR Block (1,2,3); and

   when the program requires execution of a Basic Block (2) for a given entry condition, either:

   a) if there is stored target code representing that Basic Block (2a) for that given entry condition, using said stored target code; or

   b) if there is no stored target code representing that Basic Block for that given entry condition, generating a further IR Block (2b) representative of that Basic Block for that given entry condition.

4. The method of any preceding claim, comprising:

   at the first iteration of a particular program code instruction having a plurality of possible effects or functions, generating and storing special-case intermediate representation (2a) representing only the specific functionality required at that iteration;

   at each subsequent iteration of the same program code instruction, determining whether special-case intermediate representation (2a) has been generated for the functionality required at said subsequent iteration; and

   generating an additional special-case intermediate representation (2b) specific to that functionality if no such special-case intermediate representation has previously been generated.

5. The method according to claim 4, wherein when said special-case intermediate representation (2a, 2b) is generated and stored, an associated test procedure is generated and stored to determine on subsequent iterations of the respective program code instruction whether the required functionality is the same as that represented by the associated stored special-case intermediate representation (2a,2b), and where additional special-case intermediate representation (2b) is required an additional test procedure associated with that special-case intermediate representation (2b) is generated and stored with that additional special-case intermediate representation (2b).

6. The method according to claim 5, wherein the additional special case intermediate representation (2b) for a particular program code instruction and the additional associated test procedure is stored at least initially in subordinate relation to any existing special-case intermediate representation (2a) and associated test procedures stored to represent the same subject instruction, such that upon the second and subsequent iteration of a program code instruction determination of whether or not required special-case intermediate representation (2a,2b) has previously been generated is made by performing said test procedures in the order in which they

were generated and stored until either it is determined that special-case intermediate representation (2a,2b) of the required functionality exists or it is determined that no such required special-case intermediate representation (2a,2b) exists in which case more additional intermediate representation (2b) and another associated test procedure is generated.

7. The method according to claim 5 or 6, wherein the intermediate representation (1,2,3) is optimised by adjusting the ordering of the test procedures such that test procedures associated with more frequently used special-case intermediate representation (2b) are run before test procedures associated with less frequently used special-case intermediate representation (2a) rather than ordering the test procedures in the order in which they are generated.

8. The method of any preceding claim, wherein said intermediate representation (1,2,3) is generated and stored for a block of program code (2) and subsequently re-used if the same block of program code (2) is later re-entered, and wherein at least one block of said program code (2) has alternative unused entry conditions or effects or functions and said intermediate representation (2a) is only initially generated and stored as required to execute that block of the program code with a then prevailing set of conditions.

9. The method according to claim 8, wherein for a given block of code to be translated, it is determined whether a previously stored intermediate representation (2a) therefore was for the same now currently prevailing set of conditions and, if not, then generating and storing additional intermediate representation (2b) as required to execute the block of code for the new now currently prevailing set of conditions.

10. A method of generating an intermediate representation of program code, comprising:

at the first iteration of a particular program code instruction having a plurality of possible effects or functions, generating and storing special-case intermediate representation (2a) representing only the specific functionality required at that iteration;

at each subsequent iteration of the same program code instruction, determining whether special-case intermediate representation (2a) has been generated for the functionality required at said subsequent iteration; and

generating an additional special-case intermediate representation (2b) specific to that functionality if no such special-case intermediate representation has previously been generated.

11. The method according to claim 10, wherein when said special-case intermediate representation (2a, 2b) is generated and stored, an associated test procedure is generated and stored to determine on subsequent iterations of the respective program code instruction whether the required functionality is the same as that represented by the associated stored special-case intermediate representation (2a,2b), and where additional special-case intermediate representation (2b) is required an additional test procedure associated with that special-case intermediate representation (2b) is generated and stored with that additional special-case intermediate representation (2b).

12. The method according to claim 11, wherein the additional special case intermediate representation (2b) for a particular program code instruction and the additional associated test procedure is stored at least initially in subordinate relation to any existing special-case intermediate representation (2a) and associated test procedures stored to represent the same subject instruction, such that upon the second and subsequent iteration of a program code instruction determination of whether or not required special-case intermediate representation (2a,2b) has previously been generated is made by performing said test procedures in the order in which they were generated and stored until either it is determined that special-case intermediate representation (2a,2b) of the required functionality exists or it is determined that no such required special-case intermediate representation (2a,2b) exists in which case more additional intermediate representation (2b) and another associated test procedure is generated.

13. The method according to claim 11 or 12, wherein the intermediate representation (1,2,3) is optimised by adjusting the ordering of the test procedures such that test procedures associated with more frequently used special-case intermediate representation (2b) are run before test procedures associated with less frequently used special-case intermediate representation (2a) rather than ordering the test procedures in the order in which they are generated.

14. The method of any preceding claim, comprising:

(i) generating a plurality of register objects for holding variable values to be generated by the program code; and

(ii) generating a plurality of expression objects representing fixed values and/or relationships between said fixed values and said variable values according to said program code.

15. A method of translating a computer program written for execution by a processor of a first type so that the program may be executed by a processor of a second type, the method including the step of:

generating intermediate representation, in accordance with the method of any preceding claim.

16. The method according to claim 15, wherein said translation is dynamic and performed as the program is run.

17. The method of claim 15 or 16, the method dynamically translating first computer program code written for compilation and/or translation and running on a first programmable machine into second computer program code for running on a different second programmable machine, said method comprising:

(a) generating said intermediate representation for a block of said first computer program code;

(b) generating a block of said second computer program code from said intermediate representation;

(c) running said block of second computer program code on said second programmable machine; and

(d) repeating steps a-c in real time for at least the blocks of said first computer program code needed for a current emulated execution of the first computer program code on said second programmable machine.

18. A method of optimising a computer program, the method comprising:

generating intermediate representation in accordance with the method of any of claims 1 to 14, and optimising said intermediate representation.

19. The method according to claim 18, wherein the method is used to optimise a computer program written for execution by a processor of a first type so that the program may be executed more efficiently by that processor.

20. A system arranged to perform the method of any preceding claim and including means for performing each step of said method.

21. The system of claim 20, wherein the system is an emulation system arranged for executing program code written for a first computer on a second computer.

22. The system of claim 21, wherein the second computer is of a type different to and not compatible with the first computer.

23. The system of any of claims 20 to 22, comprising:

a first programmable processor; and

an emulation system operable to execute program code written for a second processor on said first programmable processor, through generation of an intermediate representation said program code.

24. The system of any of claims 20 to 22, comprising:

a first programmable computer; and

an emulation system operable to execute program code written for a second computer on said first programmable computer, through generation of an intermediate representation of said program code.

25. A computer program comprising instructions for causing a computer to perform the method of any of claims 1 to 19.

26. A program storage medium comprising instructions for carrying out all of the steps of the method of any of claims 1 to 19.

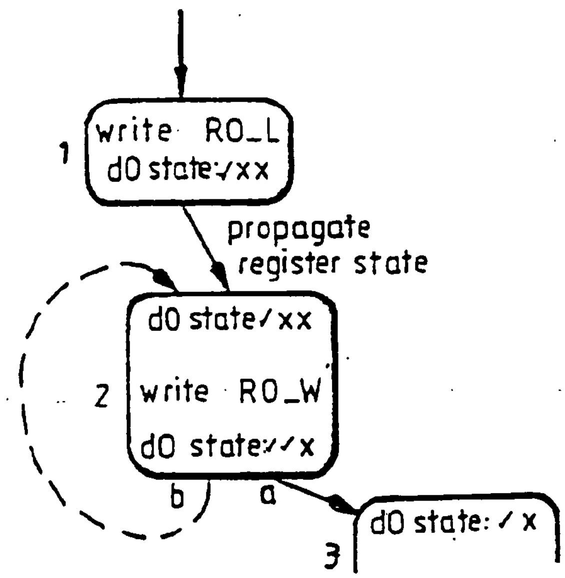
write RO_L
dO state:✓xx
1
propagate
register state
dO state✓xx
2
write RO_W
dO state:✓✓x
b
a
dO state: ✓x
3

Figure 1

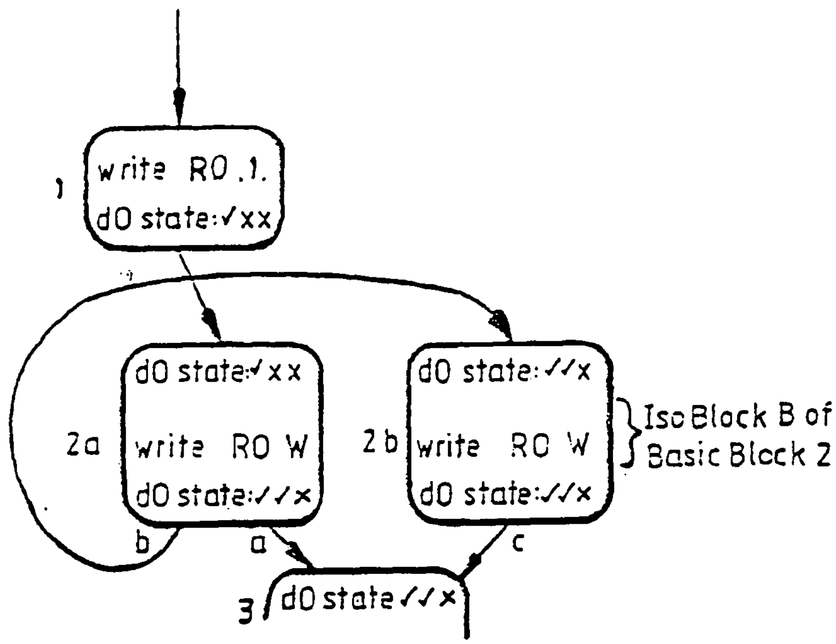
1
write R0 ,1.
d0 state:✓xx
d0 state:✓xx
2a
write R0 W
d0 state:✓✓x
d0 state:✓✓x
2b
write R0 W
d0 state:✓✓x
Iso Block B of
Basic Block 2
b
a
c
3
d0 state✓✓x

Figure 2